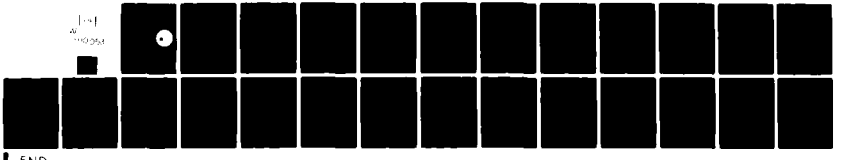
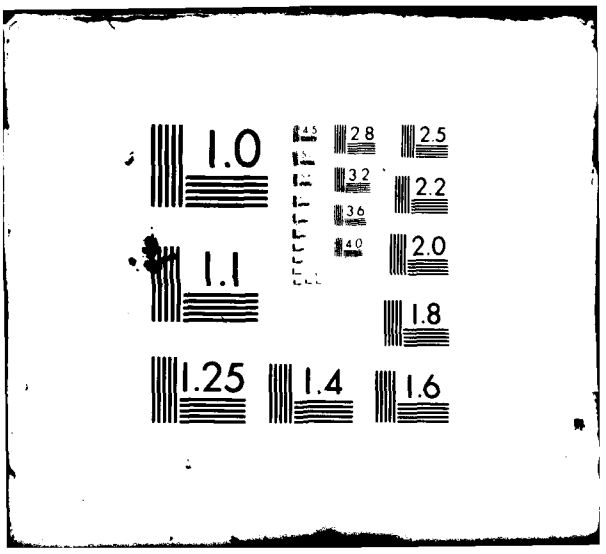
AD-A110 053 WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES F/G 9/2
A PERFORMANCE EVALUATION OF DATABASE MACHINE ARCHITECTURES.(U)
JUN 81 D J DEWITT, P B HAWTHORN DAAG29-79-C-0165
UNCLASSIFIED CSTR-437 NL

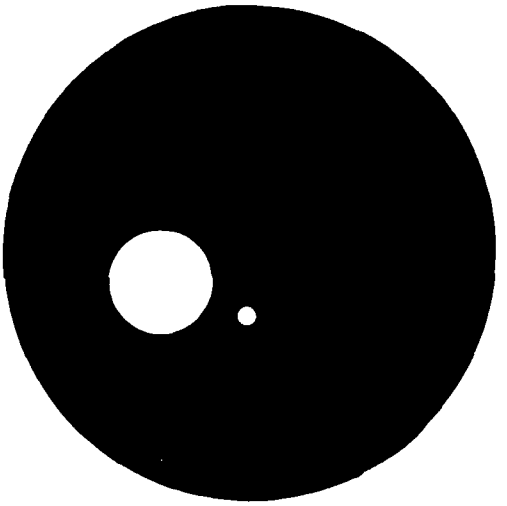


END
DATE FILMED
2 82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5

COMPUTER SCIENCES DEPARTMENT

University of Wisconsin-Madison



A PERFORMANCE EVALUATION
OF
DATABASE MACHINE ARCHITECTURES

by

David J. DeWitt
Paula B. Hawthorn

Computer Sciences Technical Report #437

June 1981

A Performance Evaluation of Database Machine Architectures

David J. DeWitt[+]
Paula B. Hawthorn[*]

[+]Computer Science Department, Univ. of Wisconsin, Madison WI.
[*]Computer Science and Mathematics Department, Lawrence Berkeley Laboratory, Berkeley, CA.

This research was partially supported by the National Science Foundation under grant MCS78-01721, the United States Army under contracts #DAAG29-79-C-0165 and #DAAG29-80-C-0041, and by the Applied Mathematical Sciences Research Program of the Office of Energy Research U. S. Department of Energy under contract W-7405-ENG-48.

ABSTRACT

The rapid advances in the development of low-cost computer hardware have led to many proposals for the use of this hardware to improve the performance of database management systems. Usually the design proposals are quite vague about the performance of the system with respect to a given data management application. In this paper we develop an analytical model of the performance of a conventional database management system and four generic database machine architectures. This model is then used to compare the performance of each type of machine with a conventional DBMS. We demonstrate that no one type of database machine is best for executing all types of queries. We also show that for several classes of queries certain database machine designs which have been proposed are actually slower than a DBMS on a conventional processor.

A

1. Introduction

The rapid advances in the development of low-cost computer hardware have led to many proposals for the use of this hardware to improve the performance of database management systems. In general, each[1] of the proposals has been quite vague about the performance of the proposed design with respect to other database machine architectures and database management systems on conventional processors. There are only two exceptions. In [OZKA77] the performance of RAP [OZKA75] is compared with that of a conventional system and in [BANJ78] the performance of the DBC [BANE78] and System R [ASTR76] are compared for selection operations. We feel that most existing database machine designs are examples of what we term "architecture directed" research. That is, database machine designers usually begin by designing what they consider to be a good architecture which they feel will efficiently execute one or two database operations. Afterwards they develop the algorithms to support all the required database operations using the basic primitives of their architecture. As an example consider associative disks (or logic-per-track devices) [SLOT70] from which RAP, RARES [LIN76], CASSM [SU75], and to some extent, DBC are derived. The basic design goal of the associative disk design was the efficient execution of the operation to select records that satisfy a certain criterion. Given this building block, other relational database operators such as join, project, and aggregate functions can be implemented with

[1] Including DIRECT [DEWI79].

varying degrees of success by combining the processing capabilities of the host with those of the back-end database machine.

In an earlier paper [HAWT80], we examined the performance of several of the proposed database management machines (associative disks, RAP, CASSM, DBC, DIRECT, and CAFS [BABB79]) with respect to several INGRES [STON76] queries. We demonstrate that no one database machine is best for executing all types of queries. For one class of queries we show that the level of performance improvement achieved does not warrant use of a database machine. While we feel that [HAWT80] represents a important first step in database machine performance comparison, it has several deficiencies that we intend to correct in this paper.

The main problem with [HAWT80] is that specific machine designs were evaluated rather then machine types. This is difficult for several reasons. First the architects of the machines always claim "foul" as they feel that the way we interpreted that their machine would process a complex query (e.g. aggregate functions) was not what they had in mind (though generally they never specified how to process such queries). A second problem with using specific machines is that many of the machines we tried to evaluate are "moving targets". Generally, database machine designers, in evaluating their architecture, find that it does not do well on a particular operation and consequently add a new wart to fix each bottleneck discovered. A third problem with comparing specific machines is that many of the proposed designs are rooted in specific, and frequently different, technologies. In attempting to compare these designs we often found ourselves

comparing "apples and oranges"

Another problem with [HATH80] is that the number of benchmark queries used was small (only 3). Furthermore, each query was applied to a small database. This makes it hard to determine whether the results obtained are representative of the performance of the machines over a range of database sizes and queries. Finally, the evaluation made no attempt to account for MIMD activity which some, but not all, of the machines can support.

In this paper, instead of simply extending the analytical model used in [HAWT80] to compare the performance of specific machines over a wider range of tests, we began by specifying five generic classes of database machine architectures which are described in Section 2. The first class is a DBMS running on a conventional computer. It is included because many of the "database machines" of the future will indeed be highly tuned DBMS software running on a single processor on top of a database operating system [GRAY78].

In Section 3, we describe our analytical performance model and introduce the various parameters that will be used to evaluate the different architecture types. Section 4 contains an evaluation of the five classes of machines on three representative query types for a variety of database sizes and queries. Our conclusions and suggestions for future research are presented in Section 5.

It is important to notice that we have completely ignored one class of proposed database machine architectures. This class contains those machines that are not feasible using today's

technology and that may never become cost effective. These machines can be spotted by claims of join times which are linear (or even less than linear) in the size of the source relations. While we are not saying that research on exotic machines is of no interest, we feel that any machine whose operation requires either as many (or more) processors as tuples in the smaller of the two relations being processed or an associative memory large enough to hold one of the relations, is a machine that will most likely never be feasible. Therefore, we ignore these machines in our performance evaluation.

2. Five Generic Classes of Database Machine Architectures

For the purpose of this evaluation, we have divided those database machine architectures which are feasible to construct using present day technology into five generic classes which we will describe in this section. These classes are:

CS - conventional systems

PPT - processor-per-track systems

PPH - processor-per-head systems

PPD - processor-per-disk systems

MPC - multiprocessor cache systems

The complexity of the database machines represented by these classes range from a conventional processor running a database operating system (CS) to a multiprocessor organization with a three level memory hierarchy (MPC). We have assumed that the last four classes of machines are connected to a host processor. This processor serves two important functions. First, it accepts and

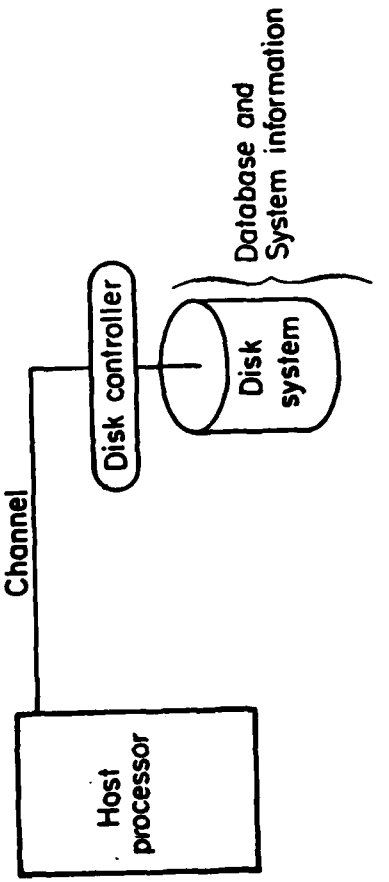
compiles queries from the users of the system. Second, depending on the functionality of the database machine, it assists in the execution of certain queries which are too complex for the back-end to handle alone. For example, the PPT, PPH, and PPD systems provide only the capability of processing selection operations. Therefore, in order to process a query which includes an aggregate function, the capabilities of the host must be used in conjunction with those of the database machine. This will be described further in Section 4.

2.1. Conventional Systems (CS)

We feel that in the future many database management system applications will be best served by a conventional relational database management system running on a single processor. Thus, our first class of "database machines" is such a system and is shown in Figure 1. We have assumed that the CS will support compiled user queries and sophisticated query execution strategies such as those employed in System R [GRIF79]. Such strategies include support of indices, sort-merge join operations, and sophisticated query optimization techniques. Furthermore, we have assumed that the operating system for the machine is tuned to the needs of a DBMS in order to minimize the overhead of I/O operations and other database activities such as locking.

2.2. Processor-per-Track (PPT) Machines

The second class of database machines are those based on a mass storage device which consists of a large number of cells. Each cell has a data track, some processing logic which can



Conventional System

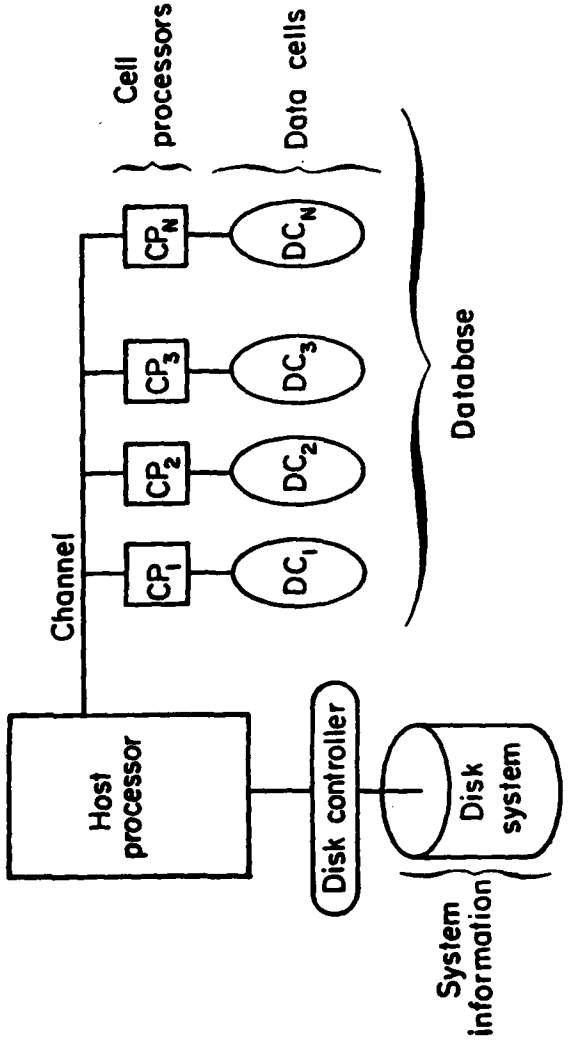
Figure 1

process selection operations "on the fly", and is connected to a global data bus on which it places selected tuples for transmission to the host processor. Coordination of the operation of the cells is performed by a controlling processor. The organization of this approach is shown in Figure 2.

In this organization tuples are stored bitwise along each track. The processing logic scans the data as the track rotates[2] and places selected tuples in a small output buffer memory associated with the head. After a buffer fills, additional logic attempts to place its contents on the output bus for transmission to the host. In the event that the processor logic is not able to output a selected tuple (because the bus is busy and the temporary storage buffers are full) processing is suspended. In this case processing will be resumed some number (1 or more) of revolutions later (i.e. after a buffer is output to the bus).

This class of machines includes the early PPT designs by Parker [PARK71], Minsky [MINS72], and Parhami [PARH72]. While the PPT architecture may appear similar to that of RAP and CASSM, the results presented in Section 4 may not be representative of these two machines (see [HAWT80] for a performance comparison of these machines). PPT machines were pioneered by Slotnick in 1970 [SLOT70] who suggested using a track of a fixed head disk as the

[2] Because of potential disk errors, the way any database machine which processes data "on the disk" must operate is to read an entire block of data into a buffer, apply a CRC, and if the block is "good" apply the selection criterion to the tuples in the block [HIBL80]. With two block buffers, loading and processing can be overlapped so that data can still effectively be processed "on the fly".



PPT System

Figure 2

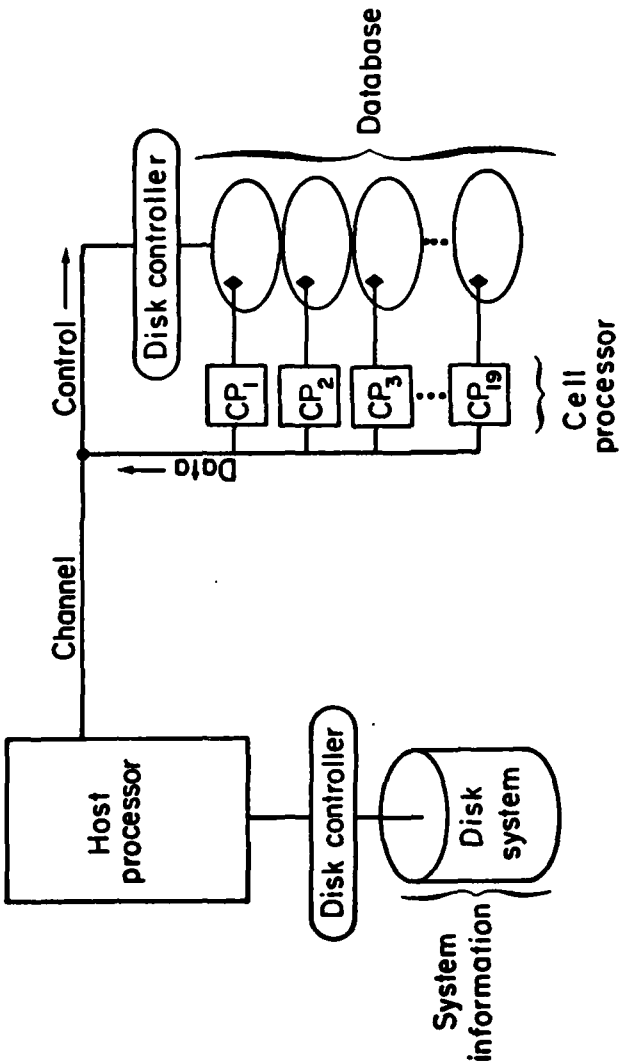
unit of storage. Since fixed head disks are being phased out of production, the reader may question whether this class of machines in indeed viable. Recent research has shown that by using magnetic bubble memories rather than fixed head disks a similar degree of functionality can be obtained with only a small reduction in performance [BORA81].

2.3. Processor-per-Head (PPH) Machines

The third class of database machines are those that associate processing logic with each head of a moving-head disk as illustrated in Figure 3. We term this class of machines "processor-per-head" machines.

In a PPH database machine, data is transferred, in parallel, over 1 bit wide data lines from the heads to a set of processors. Each processor applies the selection criteria to its incoming data stream and places selected tuples in its output buffer. In such an organization an entire cylinder of a moving head disk is examined in a single revolution (*assuming no output bus contention*). As in PPT organizations additional revolutions may be needed to complete execution of the query if an output buffer overflows.

The DBC project adopted the PPH approach over the PPT approach because PPT devices were not deemed to be cost-effective for the storage of large databases (say more than 10^{10} bytes) [KANN78]. Another possible reason for taking this route is the apparent lack of success of head-per-track disks as secondary storage devices. Moving head disks with parallel readout, on the



PPH System

Figure 3

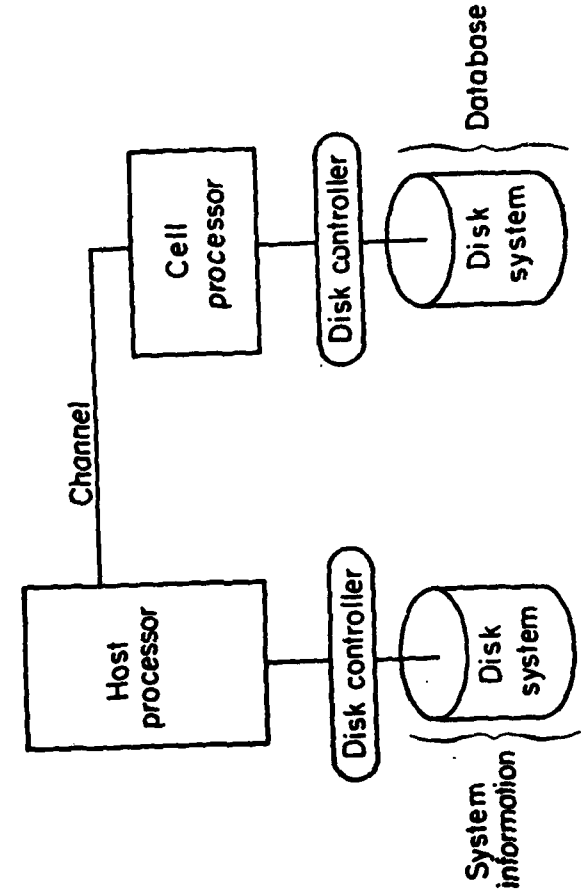
other hand, seemed an attractive and feasible alternative. The Technical University of Braunschweig in cooperation with Siemens has actually has built one for use in the Braunschweig search machine SURE [LEIL78].

2.4. Processor-per-Disk (PPD) Machines

Unlike the PPT and PPH approaches, the PPD organization utilizes a standard disk drive. In this organization a processor (or set of processors [LEIL78]) is placed between the disk and the memory device to which the selected tuples are to be transferred as shown in Figure 4. This processor acts as a filter [BANC80] to the disk by forwarding to the host only those tuples that match the selection criteria. At first glance it seems as though this approach is so inferior to the others that it does not merit any attention. However, it has the advantage that for a relatively low price one can obtain the same filtering functionality (but not the same performance) as the PPT and PPH designs.

2.5. Multi-processor Cache (MPC) Systems

The final class of database machine architectures exchange the ability to process selection operations "on-the-fly" for more functionality in the processing elements. The components of this organization are a set of small but general purpose processors and a three-level memory hierarchy. The top level of the memory hierarchy consists of the internal memories of the processors which are assumed to be large enough to hold both a compiled query and several pages of data. At the bottom level of the



PPD System

Figure 4

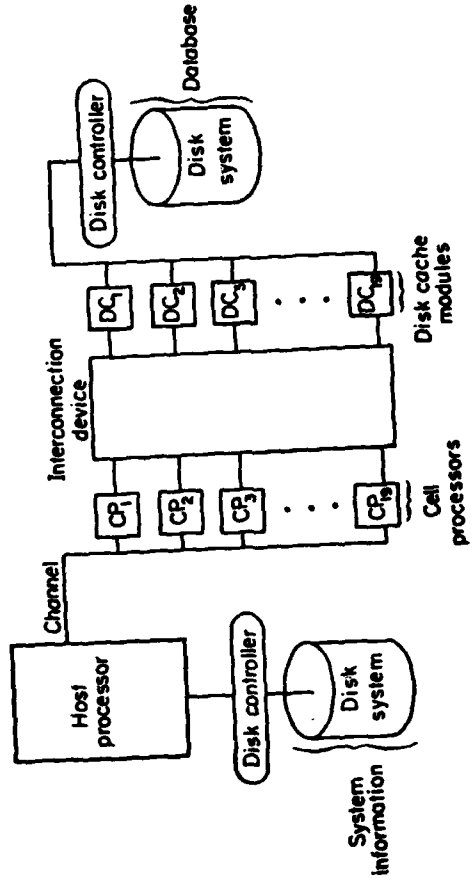
memory hierarchy are the mass storage devices used to hold the relations in the database. The middle level of the hierarchy is a disk cache which can hold one of page of data for each processor. A page of a relation is the unit of transfer between all levels of the memory hierarchy.

The bottom two levels of the memory hierarchy are connected together with a bus as shown in Figure 5. The interconnection device between the processors and the disk cache has two important properties. First it permits each processor to simultaneously read/write its block of the cache. Second it allows all processors to simultaneously read the same block of the cache (This property can also be viewed as broadcasting the page to all the processors).

There are several active database machine projects that have selected this type of architecture: DIRECT [DEWI79] which is presently operational at Wisconsin, INFOPLEX [MADN79], an MIT project has a multilevel memory processor hierarchy, the RDBM [HELL81] project at Braunschweig, and the database machine project at Texas which utilizes the TRAC [UPCH79] processor.

3. Specifications of the Database Machine Models

In this section we describe the physical and logical characteristics of the five classes of database machine architectures that we modeled.



MPC System

Figure 5

3.1. Physical Characteristics

3.1.1. Mass Storage Device Specifications

The mass storage device employed in our CS, PPD, PPH, and MPC models is based on the IBM 3330 disk drive [GORS80]. This device has 404 cylinders with 19 tracks (recording surfaces) per cylinder. Each track holds 13,030 bytes. The rotational speed of this disk drive is one revolution every 16.7 ms. The average access time to a random block, T_{DAC}, equals 38.6 ms (the time required to seek 202 cylinders and wait 1/2 revolution). The track-to-track seek time, T_{SK} is 10.1 ms. Table 3.1 summarizes these parameters.

Table 3.1 Disk parameters and values

parameter	description	value
BSIZE	block size	13,030 bytes
DCYL	# blocks/cylinder	19
T_{IO}	block read/write time	16.7 ms.
T_{DAC}	average access time	38.6 ms.
T_{SK}	track-to-track seek time	10.1 ms.

3.1.2. Conventional and Host System Specifications

The processor for the conventional system (and the host processor for the back-end systems) is assumed to be a 1 MIP processor such as a VAX 11/780. As described in the Section 2, we have assumed that the CS machine runs a relational database management system which supports compiled queries and sophisticated query optimization and execution techniques. We have also assumed that performance is enhanced through the support of a database

operating system in order to minimize overhead costs. Finally, we have assumed that a slightly modified version of the same system is used as the host processor for all the backend systems. The parameters which characterize the operation of this system are presented below in Table 3.2.

The values presented in Table 3.2 represent a combination of measurements performed on INGRES [HAWT79], System R [CHAM81], and some "back of the envelope" calculations. T_{SC} represents the time to perform a simple scan (e.g. a selection operation) on a block of data. The time required for such an operation is obviously dependent on such factors as tuple length and query type (e.g. whether the type of the attribute being compared is a string or integer). If there are 130 tuples in each block, then 10 ms/block permits approximately 75 instructions for processing each tuple. The time required to perform a complex operation such as internally sorting a page or merging two pages in order

Table 3.2 CS parameters and values

parameter	description	value
T_{SC}	cpu time to scan the tuples in a block	10.0 ms.
T_{BLK}	cpu time for a complex operation on a block	95.0 ms.
T_{OIO}	cpu time to initiate an I/O operation	2.0 ms.
T_{CODE_GEN}	cpu time to compile a query	152.0 ms.
T_{INDEX}	time required to fetch and examine an index page	67.0 ms.
T_{MSG}	cpu time to send/receive a message from the back-end	2.0 ms.

to perform a merge sort is represented by T_{BLK}. [KNUTH75] shows that for a page containing k tuples at most 2k tuple comparisons and moves are required to merge two sorted pages. Thus, for 130 tuples/block, 95 ms. appears to be a realistic estimate for T_{BLK}.

Even though we have assumed that a database operating system will minimize system overhead, some time must be associated with processing an I/O request or sending a message to the database machine. The overhead associated with these two operations is represented by T_{OIO} and T_{MSG} respectively.

Finally, a time must be associated with compiling a query, T_{CODE_GEN}, and processing an index T_{INDEX}. We have assumed that compiling a query requires at least one I/O operation (with its associated overhead) and an amount of cpu time equivalent to T_{BLK}. Examining an index is assumed to require one[3] I/O operation to fetch the appropriate block of the index from disk plus an amount of cpu time equivalent to T_{SC}.

3.1.3. PPD, PPH, and PPT Specifications

For the PPD, PPH, and PPT database machine designs we have assumed that the processors compare a data stream from the disk with another data stream that contains the query which has been compiled into a format compatible with that of the disk data stream. Selected tuples are saved in a small output buffer for transmission over a common bus to either a host or controlling processor. We have assumed that a processor is fast enough to process the selection operation at the speed of the incoming data

[3] A figure of 2 or 3 is probably somewhat more accurate

stream. Thus, the time for a PPT, PPH, or PPD processor to process a block is T_{IO}. For most conventional disks a processor has approximately 1.25 microseconds to process each incoming byte. Assuming that it takes 3 instructions to examine a byte (1 instruction for byte comparison with auto-increment, 1 to test for loop termination, and 1 for the branch instruction) and that every byte must be examined, then each processor for the PPT, PPH, and PPD designs must be approximately a 2.4 MIP processor.

The PPD database machine design was modeled as one IBM 3330 disk drive and one processor. This design requires T_{IO} ms. to process each track (block) occupied by the relation being searched.

The PPH database machine was modeled as a modified IBM 3330 disk drive with 19 processors (one per head) and two output buffers per head. Each output buffer is capable of holding one tuple.[4] Since this design has a processor associated with each head, it can process an entire cylinder (19 blocks) in T_{IO} ms.

In order for the PPT design to have similar storage capacity and performance characteristics as the PPH and PPD designs, we modeled it as a 3330 disk drive with one processor for each of the 7676 tracks (404 cylinders * 19 tracks/cylinder) and two output buffers per head. While constructing such a device is probably out of the question, modeling the PPT design this way enables us to establish a performance baseline by which the performance of the other database machines can be gauged. Since

[4] [BORA81] has shown that the size of the buffers has minimal impact on performance.

there is one processor per track, the entire disk can be searched in T_{IO} ms. if there is no contention among the processors for access to the output channel.

3.1.4. MPC Specifications

The MPC database machine was modeled as nineteen[5] 1 MIP processors, a cache consisting of nineteen blocks of RAM, and one standard IBM 3330 disk. Since the processors have the same performance as the processor used in the CS, the values for T_{SC} and T_{BLK} remain the same. However, before the processors can examine the contents of a cylinder of the disk, one block must be moved to the local memory of each processor. This operation requires three steps. First the heads must be positioned to the proper cylinder. This step requires either T_{DAC} or T_{SK} ms depending on the previous position of the heads. Second, the contents of the cylinder must be transferred to the disk cache. This requires $19*T_{IO}$ ms. Finally, a parallel read is performed by the processors requiring an additional T_{IO} ms. Selected tuples are placed on the bus connecting the processors to the host processor.

3.1.5. Output Channel Specifications

As discussed in Section 2, all processors for the PPT, PPH, PPD, MPC were assumed to be connected to a single output channel for the transfer of results to the controlling or host processor. We assumed that this output channel operated independently and asynchronously from the cell processors. The bandwidth of this

[5] We selected 19 processors simply because that is how many processors the PPH design has.

channel was assumed to be 2.0 Mbytes/second based on the maximum bandwidth of the VAX 11/780's Mass Bus Adapter. Thus, the transfer time for one block, T_{BT}, is 6.5 ms. It should be noted that the output channel must be as fast as the disk data transfer rate, although it can be faster. The disk transfer rate determines the processor speed for the processors employed in the PPT, PPH, and PPD designs, while the output channel bandwidth affects the rate at which output buffers in the processors will be emptied (recall that loading and unloading of the buffers are asynchronous operations).

3.2. Operational Characteristics

For the PPD, PPH, and MPC database machine designs relations are stored in such a manner as to occupy the minimum number of cylinders possible. That is, tuples from a relation must first fill an entire track before a second track is used, then an entire cylinder, etc. In this way, the number of cylinders which must be searched to execute a selection operation on a relation is minimized and non-essential seek operations are eliminated. As first suggested by Sadowski and Schuster [SADO78], concurrency can be maximized in the processing of a selection operation in a PPT database machine if tuples from a relation are uniformly distributed across all tracks.

4. Performance Comparisons

In this section we benchmark the five database machine designs on selection, join, and aggregate function operations. These operations were chosen as each is representative of a class

of query types. Selection queries are representative of those types of queries which can be performed in O(n) time for n tuples on a single processor. Deletion, which can be viewed as a negated selection, can also be performed in one pass through the relation. The performance of the machines executing join queries is representative of those operations (e.g. division) involving two relations and requiring either O(nlogn) or $O(n^2)$ time on a single processor. Finally, aggregate function queries serve as a benchmark for complex operations which reference a single relation yet require O(nlogn) time on a single processor (e.g. projection with duplicate elimination).

The performance evaluations which are presented below measure the total system work necessary to process a query not the response time of the database machine to the query. It is certainly the case that certain operations can be overlapped. However, determining (and explaining) exactly which processing steps can be overlapped significantly complicates the descriptions below. Furthermore, total system work is probably a more accurate measure of the resources consumed by a query than response time.

In this section the number of disk blocks occupied by a relation R will be denoted |R|. The selectivity factor of selection operations is denoted by f.

4.1. Selection Queries

The processing of a selection operation can be divided into two cases depending on whether a secondary index exists on the attribute being qualified. We have assumed that the existence of an index on the attribute being qualified reduces both the number of cylinders that must be searched (from |R|/19 to |R|*f/19[6]) and the number of blocks that must be processed (from |R| to |R|*f). Since the performance characterization of each system in the presence of indexing is a straightforward modification of the non-index case, they have not been included.

4.1.1. CS

Processing a selection operation on a conventional system involves three main components: query compilation, positioning the heads to the proper cylinder, and processing each block occupied by the relation. Query compilation requires T_{CODE_GEN} ms. Head movement requires T_{DAC} for the initial head movement and one track-to-track seek (T_{SK}) for each additional cylinder. The time to process each block occupied by R consists of the I/O time to read the block, T_{IO}, the cpu time required by the operating system to handle the I/O operation, T_{OIO}, and the cpu time required to apply the selection operation to the block, T_{SC}. We have ignored the time to display the results of the query as this will be very device dependent.

$$T_{SEL\_CS} = T_{CODE\_GEN} + T_{DAC} + ((|R|/19)-1)*T_{SK} + |R|*(T_{IO}+T_{OIO}+T_{SC})$$

[6] Recall that there are 19 blocks/cylinder.

4.1.2. PPT

Execution of this query on the PPT machine will begin with the host compiling the query, T_{CODE_GEN}, and then sending the compiled query to the database machine for execution. Sending the query is assumed to take T_{MSG} ms. of host time in the form of operating system overhead. Since the PPT machine can examine every block in one revolution of the disk, the qualifying tuples will be located in one revolution. However, the execution time will be the maximum of the time for one disk revolution and the time to return all but the last block of selected tuples to the host. The time to return each block of selected tuples consists of two components: the time, T_{BT}, to send each block to the host as determined by the transfer rate of the bus and the cpu time consumed by the operating system to process each block returned, T_{MSG}. The time to process the query is thus:

$$T_{SEL\_PPT} = T_{CODE\_GEN} + T_{MSG} + MAX\{T_{IO}, (\lceil |R|*f \rceil - 1)*(T_{MSG}+T_{BT})\} + (T_{MSG}+T_{BT})$$

Since the whole database can be searched in one revolution, an index will not reduce the execution time of the query.

4.1.3. PPH

Processing this query is similar to the PPT design except that processing each cylinder occupied by the relation requires T_{IO} ms. In addition, the time to seek to the first cylinder occupied by the relation and the track-to-track seek times must be incorporated. The execution time is thus:

$$T_{SEL\_PPH} = T_{CODE\_GEN} + T_{MSG} + T_{DAC} + MAX\{(\lceil |R|/19 \rceil - 1)*T_{SK} + \lceil |R|/19 \rceil * T_{IO}, (\lceil |R|*f \rceil - 1)*(T_{MSG}+T_{BT})\} + (T_{MSG}+T_{BT})$$

4.1.4. PPD

Execution on PPD machines will be similar to the PPT and PPH designs except that now one revolution will be required for each block (track) occupied by the relation referenced. Therefore, the execution time for the PPD design will be:

$$T_{SEL\_PPD} = T_{CODE\_GEN} + T_{MSG} + T_{DAC} + MAX\{(\lceil |R|/19 \rceil - 1)*T_{SK} + |R|*T_{IO}, (\lceil |R|*f \rceil - 1)*(T_{MSG}+T_{BT})\} + (T_{MSG} + T_{BT})$$

4.1.5. MPC

Although the MPC design has 19 processors which can be used to process selection queries, in order for the blocks of R to be processed they must first be moved into the disk cache. Once there, processing the blocks from the cylinder takes two steps. First the processors each read (in parallel) a block from the cache. This step will require T_{IO} ms. The second step is for each processor to apply the selection condition of the query to extract the qualified tuples. This step will require T_{SC} ms. Since processing one cylinder can be overlapped with loading the contents of the next cylinder into the cache, the processing time of the query can be modeled as the time required to move all cylinders containing a block of R into the cache plus the processing time of the last cylinder. As with the other designs, we

have assumed that processing can be overlapped with transmission of the selected tuples to the host. Therefore, the execution time for the MPC design is:

$$T_{SEL\_MPC} = T_{CODE\_GEN} + T_{MSG} + T_{DAC} + MAX\{(\lceil R\rceil/19\rceil-1)*T_{SK}+\lceil R\rceil*T_{IO}+T_{IO}+T_{SC}, (\lceil R\rceil*f\rceil-1)*(T_{MSG}+T_{BT})\} + (T_{MSG} + T_{BT})$$

4.1.6. Evaluation

Using these formulas we evaluated the performance of each of the database machine designs for a number of different relation sizes over a range of selectivity factors. The results from one experiment for a relation consisting of 50,000 one-hundred byte tuples are presented in Tables 1 and 2 below. Table 1 contains the performance of each machine for four different selectivity factors when no index exists on the attribute being qualified. Table 2 presents their performance when an index does exist.

These tests indicate some interesting results. First, we

Table 1
50,000 Tuples of Size 100 bytes
No Index Case

Selectivity Factor of Query	CS	Execution Time in Seconds PPT	PPH	PPD	MPC
.0001	11.398	.179	.751	6.801	6.818
.001	11.398	.179	.751	6.801	6.818
.01	11.398	.188	.751	6.801	6.818
.1	11.398	.486	.751	6.801	6.818

Table 2
50,000 Tuples of Size 100 bytes
Index Case

Selectivity Factor of Query	CS	Execution Time in Seconds PPT	PPH	PPD	MPC
.0001	.287	.179	.285	.285	.312
.001	.287	.179	.285	.285	.312
.01	.373	.188	.294	.335	.362
.1	1.396	.486	.592	.939	.965

feel that if the complexity of the PPT design is considered, it is not a cost-effective design even when evaluated on the operation for which it was specifically designed. Its performance is superior (a factor of 5) to the PPH design for the case of a low selectivity factor and no appropriate index. In the remaining cases the PPH design is almost as fast. It is especially important to notice how the performance of the PPT design degrades with higher selectivity factors (.01 to .1) due to contention for the channel to the host. Another interesting observation is the very good performance of the CS and PPD designs when an index exists on the attribute being qualified. If access to a relational database is such that the appropriate indices can always be maintained the CS and PPD designs are undoubtably the most cost-effective for processing selection queries. If the appropriate indices cannot always be maintained than the overall performance of the PPH design makes it the most reasonable choice.

4.2. Join Queries

While each of the database machines processed selection queries in a basically the same manner, the algorithms used for processing join queries are very different. Let R and S be the two relations to be joined. |R| and |S| will denote the number of disk blocks occupied by both relations. The length of the tuples in S is represented by S_len.

4.2.1. CS

The join algorithm used by the CS design is a sort-merge algorithm that was shown to be one of the best join algorithms for a single processor system in [BLAS77]. The first step of this algorithm is to sort both relations on the joining attribute (assuming neither is already sorted on the appropriate attribute). We have assumed that the two relations are sorted using a 4 way external merge sort algorithm. Execution of the algorithm requires $\log_4(X)$ phases. During each phase all pages are read and written (hence the factor of 2) and approximately X/2 two-page merge operations are performed. The time to sort a relation with comprised of X pages is

$$\log_4(X)*2*\{(T_{DAC}+([X/19]-1)*T_{SK}+X*(T_{IO}+T_{OIO})\} + \log_4(X) * (X/2)*T_{BLK}$$

The second phase of the sort-merge join algorithm is to merge the two sorted relations emitting tuples that satisfy the join condition. We have also assumed that the merge step of the algorithm can be performed by reading each block of both sorted relations exactly once. While this assumption is generally

valid, there are extreme cases (ie. all tuples in both relations have the same join attribute value) when one of the sorted relations must be repeatedly read. Finally note we have assumed that the sorted relations reside on different cylinders of the disk so that each time a block of one of the relations is read, the time for an "average" disk access, T_{DAC}, is accrued.

The time to execute this algorithm is:

$$T_{JOIN\_CS} = T_{CODE\_GEN} + T_{SORT\_R} + T_{SORT\_S} + T_{MERGE}$$

where

$$T_{MERGE} = (|R|+|S|)*(T_{DAC} + T_{IO} + T_{OIO}) + (|R|+|S|)*T_{BLK}/2$$

4.2.2. PPT, PPH, and PPD

Since the only functionality provided by the PPT, PPH, and PPD designs is to process selection queries, each of these designs process queries by decomposing the query using an algorithm based on Wong's [WONG76] tuple substitution algorithm. Assume that the join operation as specified by the user has the form R.a = S.b and that S contains fewer tuples than R. These database machine designs will process the query by issuing one selection subquery for each tuple in S. The form of this subquery will be R.a = x where x is the join attribute value from the current tuple in S. The result relation is produced by "joining" each tuple in S with all the tuples from R returned by the execution of its subquery.

The following formulas express the join execution time for the PPT, PPH, and PPD designs assuming that relation S contains

fewer tuples than R. The first step is for the host to compile the query. Next, for each block in S, the host will read the block ($T_{DAC} + T_{OIO} + T_{IO}$) and then use the database machine to execute a subquery for each tuple in the block. We have included a factor of T_{BLK} in the cost of processing each block of S to reflect the cost of processing each subquery (finding the next tuple in S and forming the result relation). Although we have used the notation T_{SEL_PPD} (for example) to express the cost of performing each subquery, for the results presented in Tables 3 and 4 the cost of code generation was not included in the cost of executing each subquery.

$$T_{JOIN\_PPT} = T_{CODE\_GEN} + |S|*[T_{IO}+T_{OIO}+T_{BLK} + (BSIZE/S\_LEN)*T_{SEL\_PPT(R)}]$$

$$T_{JOIN\_PPH} = T_{CODE\_GEN} + |S|*[T_{DAC} + T_{IO} + T_{OIO} + T_{BLK} + (BSIZE/S\_LEN) * T_{SEL\_PPH(R)}]$$

$$T_{JOIN\_PPD} = T_{CODE\_GEN} + |S|*[T_{DAC} + T_{IO} + T_{OIO} + T_{BLK} + (BSIZE/S\_LEN) * T_{SEL\_PPD(R)}]$$

If the query issued by the user contains a selection on R and S followed by a join of the restricted relations, then the query can be executed in two steps. First one of the selections is executed (the host should choose the one which will produce the smaller result relation). For each tuple in the resulting relation a complex selection subquery combining the join condition with the selection condition of the second relation will be executed.

4.2.3. MPC

The join algorithm used by the MPC design is a block parallel version of the nested-loops algorithm. Execution of this algorithm begins by having each processor read a different page of R. Next all pages of S are sequentially broadcast to the processors. As each page of S is received by a processor it joins the page with its page from R using a 2 way "merge".[7] Since the number of processors available (19 for MPC) is generally less than |R|, this process is repeated [|R|/19] times. Thus,

$$T_{JOIN\_MPC} = T_{CODE\_GEN} + T_{MSG} + MAX(T_{EXECUTE\_JOIN}, T_{SEND\_RESULTS}) + (T_{MSG} + T_{BT})$$

where:

$$T_{EXECUTE\_JOIN} = \lceil |R|/19 \rceil (T_{DAC} + 19*T_{IO} + T_{IO} + T_{DAC} + (\lceil |S|/19 \rceil - 1)*T_{SK} + 2*T_{IO} + |S|*T_{BLK})$$

$$T_{SEND\_RESULTS} = ((\lceil |R|*|S|*jsf \rceil - 1) * (T_{MSG}+T_{BT})$$

In the formula for $T_{EXECUTE\_JOIN}$, there are [|R|/19] phases. Each phase begins by having the processors read the next 19 pages of R. This requires $T_{DAC} + 19*T_{IO} + T_{IO}$ ms (the final T_{IO} is for the parallel read). Next all of S must be joined with the current 19 pages of R. Since $T_{BLK} > T_{IO}$, reading pages of S can be overlapped with their processing except for an initial $2T_{IO}$ ms. period. in which the first page of S is transferred first to the cache and then broadcast to the processors. In the formula for $T_{SEND\_RESULTS}$, jsf represents the join selectivity factor and

[7] We have assumed that the blocks of R and S are internally sorted on the join attribute. See [BORA80] for a description of parallel update algorithms that always leave blocks sorted.

thus |R|*|S|*jsf is the number of pages in the result relation.

4.2.4. Evaluation

We quantified the join algorithm performance of the five machines through a number of different tests. In Tables 3 and 4 below, the result of one of those tests are presented. Table 3 presents the performance of each system when an index on the join attribute for R does not exist. The performance of the PPH and PPD designs when an appropriate index exists is illustrated in Table 4. For the results presented in Table 4, we assumed that only 1 out of every 10 subqueries incurred the cost of processing an index request (T_{index}).

We found these results very interesting for several reasons. First is the terrible performance of the PPT, PPH, and PPD designs. These results make it clear that, even if the appropriate index exists, that "tuple substitution" is not a reasonable

Table 3
Relation R: 10,000 Tuples of Size 100 bytes
Relation S: 3,000 Tuples of Size 75 bytes
No Index Case

Selectivity Factor of Query	Execution Time in Seconds				
	CS	PPT	PPH	PPD	MPC
.0001	34.4	83.7	520.2	4120	7.7
.001	34.4	83.7	520.2	4120	7.7
.01	34.4	83.7	520.2	4120	7.7
.05	34.4	83.7	520.2	4120	7.7

Table 4
Relation R: 10,000 Tuples of Size 100 bytes
Relation S: 3,000 Tuples of Size 75 bytes
Index Case

Selectivity Factor of Query	Execution Time in Seconds	
	PPH	PPD
.0001	220.5	220.5
.001	220.5	220.5
.01	220.5	220.5
.05	247.2	370.5

way of processing joins. The consequence is that if the ability to perform selections rapidly is the only functionality provided by a database machine, then it is better to ignore the database machine and do a sort-merge join on the host.

Because we were puzzled about the very poor performance of the PPT, PPH, and PPD designs (and expect that the reader is also), we have broken apart the processing costs in Table 5 for a join selectivity factor of .0001. The existence of the appropriate index has been assumed for the PPH and PPD designs. For all three designs, the processing time is dominated by the time to process the subqueries. The time in the PPH and PPD designs which is devoted to processing subqueries (75% of the total), is composed of 70% seek time and 30% processing time. While this 70% figure seems high and consequently might be an area for optimization, one only need to look at the performance of PPT design (which obviously has no seek time) to see that such efforts would be fruitless.

While the performance of the MPC design clearly indicates

Table 5

Selectivity Factor = .0001

Activity	Percentage of Effort PPT	PPH	PPD
Query Compilation	.18%	.07%	.07%
Reading Relation S	3.3%	1.2%	1.2%
Processing R Index	--	9.2%	9.2%
Sending Subquery Messages	7.2%	2.7%	2.7%
Processing Subqueries	59.7%	75.2%	75.2%
Handling Subquery Result Messages	30.4%	11.6%	11.6%

the need for general purpose processors in the back-end database machine, the results are disappointing. Notice that although the MPC had 19 times as many processors as the CS, the speed-up factor was only 4. Other experiments indicated, however, that the speed-factor achieved by the MPC design ranges depending on the operand sizes from 1 to 4 with the average being approximately 2. In fact, a speed factor of 4 is only achieved when one of the relations has 19 or fewer pages

We initially speculated that the relatively poor performance of the MPC design was due to a mismatch in processor performance and I/O bandwidth. To test this hypothesis, we modified the MPC equation to utilize a disk drive with parallel readout. The results were very surprising as the performance of the MPC design improved by only 3.5%. Further analysis of the join execution time for the MPC design revealed that the execution time of the

algorithm is dominated by reading, broadcasting, and processing pages of the inner relation. The performance of this sequential operation cannot be improved by the addition of parallel I/O hardware. In fact the only step of the parallel nested loops algorithm in which parallel I/O can be exploited is reading the pages of the outer relation.

4.3. Aggregate Function Queries

There are two basic types of aggregate queries supported by relational database systems such as INGRES: "scalar" aggregates and aggregate "functions". Scalar aggregates are aggregations (average, max, etc.) over an entire relation. Aggregate functions first divide a relation into non-intersecting partitions (based on some attribute value, e.g. sex) and then compute scalar aggregates on the individual partitions. Thus, given a source relation, scalar aggregates compute a single result while aggregate functions produce a set of results (i.e. a result relation). In this section we examine the performance of each machine on aggregate function queries.

4.3.1. CS

The algorithm used by the CS design for executing aggregate function queries is to first sort the relation on the partitioning attribute(s) in order to bring all tuples in the same partition together. Next a scan is made of the entire relation and the value for each partition is computed. Two easy optimizations are possible. Assume that a 4 way external merge sort is used to sort R. Then, the last phase of the merge sort will be begin with 4

runs of length |R|/4. Instead of finishing the sort during this last phase, the value for each partition can instead be computed. The second optimization is that the result of the final step of the sort can be thrown away instead of writing it to disk. The cost of this algorithm is:

$$T_{AGG\_CS} = T_{CODE\_GEN} + T_{SORT\_R} + T_{FORM\_RESULT}$$

where:

$$T_{SORT\_R} = (2*\log_4|R|-1)*\{T_{DAC}+([R/19]-1)*T_{SK}) + |R|*(T_{OIO}+T_{IO})\} + \log_4|R|*(|R|/2)*T_{BLK}$$

$$T_{FORM\_RESULT} = |R|*T_{BLK}$$

4.3.2. PPT, PPH, and PPD

These three database machines execute aggregate functions in two steps. First the host projects the source relation on the partitioning attribute(s) to form a list of partition identifiers using an external merge sort on the partitioning attribute.[8] This step causes the source relation to be reduced in size from |R| blocks to |N| blocks. If the size of the projected tuples is PT_LEN, then the number of partitions, P, is |N|*BSIZE/PT_LEN. Next for each tuple in the projected relation a subquery is sent to the database machine. As the selected tuples are returned from each subquery the scalar aggregate operation is applied to compute the value for the partition. Since each tuple in R will be selected by one and only one subquery, the cpu time for

[8] See Appendix A for a derivation of $T_{PROJECT}$. For the evaluation presented below we have assumed a four way merge sort.

computing the aggregate partition values can be estimated to be $|R|*T_{BLK}$. Thus the execution time for this query is:

$$T_{CODE\_GEN} + T_{PROJECT} + T_{SUBQUERIES} + |R|*T_{BLK}$$

where:

$$T_{PROJECT} = \log_4|N|*\{2*T_{DAC}+2*|R|*(T_{IO}+T_{OIO})+|R|*T_{BLK}) + 2*([|R|/19]-1)*T_{SK}\} + \log_4(|R|/|N|)*2*T_{DAC} + \{(|R|-|N|)/3\}*\{4*(T_{IO}+T_{OIO}+T_{BLK}+T_{SK}/19) + (T_{IO}+T_{OIO}+T_{SK}/19)\}$$

For the PPT design, $T_{SUBQUERIES} = P * T_{SEL\_PPT}(R)$

For the PPH design, $T_{SUBQUERIES} = P * T_{SEL\_PPH}(R)$

For the PPD design, $T_{SUBQUERIES} = P * T_{SEL\_PPD}(R)$

4.3.3. MPC

In the first stage of the MPC algorithm, each processor reads a set of source relation blocks and accumulates one aggregate value for each partition it sees (at most P). This results in a number of blocks containing partial results which must be combined. The second stage is a parallel "merge" of the blocks produced in the first stage to produce the final result. The cost of this algorithm may be computed as:

$$T_{AGG\_MPC} = T_{CODE\_GEN} + T_{MSG} + T_{EXECUTE} + T_{RETURN\_RESULTS}$$

where:

$$T_{EXECUTE} = T_{FORM\_PARTIAL\_RESULTS} + T_{PARALLEL\_MERGE}$$

The time to form the partial result blocks consists of a component to process R, $T_{PROCESS\_R}$, and a component to write the partial result blocks to mass storage, T_{PR_IO}. Processing R begins by having each processor read a separate block of R. Each

tuple in the block must be placed in the correct partition and the aggregate value for that partition must be updated. If the partitions are kept in sorted order based on the partitioning attribute(s), then a binary search can be used to locate the correct partition. However, when a new partition is seen it must be placed in its proper place. T_{BLK} ms. seems to be a reasonable estimate of the cost of processing the tuples in a block (including keeping the partitions in sorted order). Since |R| is generally greater than 19 (the number of processors), this step will be repeated [|R|/19] times. Thus,

$$T_{PROCESS\_R} = T_{DAC}+\{(|R|/19)*\{T_{SK}+19*T_{IO}+T_{IO}+T_{BLK}\}$$

If we assume that each processor sees every partition, then the first step of the first phase will produce 19*|N| blocks. The cost of writing these blocks to mass storage before the parallel merge begins is:

$$T_{PR\_IO} = T_{DAC} + (|N|-1)*T_{SK} + |N|*19*T_{IO}$$

The parallel "merge" we use in the second stage is not a true merge since two partial result blocks are combined to form a single result block. The parallel merge occurs in two steps. First, each processor using a 2-way merge sort must form a sorted run of the |N| blocks it has produced. The time required for this step is:

$$2*\log_2|N|*\{T_{DAC} + (|N|-1)*T_{SK} + |N|*(19*T_{IO} + T_{IO})\} + \log_2|N|*|N|*T_{BLK}$$

Next a pipelined-parallel binary merge (see [BORA80]) is

used to combine the 19 runs of |N| blocks into one run of |N| blocks. The number of stages used is $\log_2(19)$. Each processor will read two runs of |N| pages, merge them, and write a run of length |N|. The cost of the parallel "merge" is:

$$T_{DAC}+(|N|-1)*T_{SK}+|N|*19*T_{IO}+(|N|+\log_2 19)*(2*T_{IO}+T_{BLK}+T_{IO})$$

Finally, the cost of returning the results to the host is:

$$T_{RETURN\_RESULTS} = |N|*(T_{MSG} + T_{BT})$$

4.3.4. Evaluation

The results from evaluating the five designs for a variety of different partition sizes on a relation with 50,000 tuples is shown in Table 6 below. As indicated by these results, for up to 50 partitions the subquery approach appears viable for the PPT and PPH designs. It is also interesting to note that the MPC design achieves a speed-up factor of 17:1 over the conventional system up to 250 partitions. This seems to indicate that as the operation to be performed becomes more complex, the impact of serial I/O from the disk is decreased. The performance of the MPC algorithm degrades for a large number of partitions because we assumed that each processor would see a tuple from every partition. Thus, for 5,000 partitions, the result of the first phase of the algorithm are 19 partial result relations with 39 blocks each. This relation is twice the size of the original relation.

Table 6
50,000 Tuples of Size 100 bytes

Number of Partitions	Execution Time in Seconds				
	CS	PPT	PPH	PPD	MPC
5	171.1	101.0	101.2	131.0	10.2
50	171.1	101.2	127.7	430.1	10.2
250	171.1	130.0	273.0	1785.	10.3
500	171.1	162.1	448.0	3473.	19.3
2500	171.1	273.3	1703.	16828	89.9
5000	171.1	363.0	3223.	33471	193.2

5. Conclusions

From these experiments we are able to draw a number of conclusions about these five database machine designs. First, for processing selection operations the PPH design is probably the most cost effective. While the CS, PPD, PPH, and MPC designs all process selection queries very efficiently when an index exists on the selected attribute, only the PPH design can provide a reasonable level of performance when an index does not exist. If the intended application of the database machine is such that an appropriate index can always be maintained then the PPD design is clearly the winner.

For complex queries, the MPC design with its general purpose processors was demonstrated to be the fastest design. However, the design we presented clearly has a number of drawbacks. First, as indicated by the rather poor speed-up achieved for

executing some operations, the effective processor utilization is very low. As indicated previously the problem seems to stem at least partially from a mismatch between processor performance and I/O bandwidth. This lack of sufficient I/O bandwidth also manifests itself in the performance of the MPC when processing selection queries without the appropriate index.

These results seem to indicate several avenues of future research. One obvious area that might be worthwhile to investigate is to design a machine which combines the ability of the PPH design to process selection queries "on the fly" with the ability of the MPC design to process complex queries. Such an approach may or may not be the solution we are all seeking. For example, maybe the solution instead is to simply use a parallel read out disk as the mass storage device for the MPC design. Another, possibly fruitful, approach is to look at a database machine design in which each active query is run on a separate processor using the same algorithms presented for the CS. This is the approach taken by the MUFFIN project [STON79]. While this approach looks appealing, it may result in problems with data placement and poor processor utilization.

Rather than advocate either approach, we suggest a third approach. This begins with a careful examination and analysis of the algorithms which the architect intends for his database machine to execute.[9] We feel that in order to develop a truly efficient and cost effective machine one must first develop the

[9] See [BORA80] for one such attempt

algorithms and then extract the primitive operations which are necessary for an efficient implementation of the algorithms. Only after these primitives are known and understood, should one attempt to design a machine.

Finally, we would like encourage people to use our analytical approach to benchmark their favorite database machine. While this approach may not be perfect, we hope that this paper has demonstrated that it can produce useful results.

6. Acknowledgements

The authors would like to thank Haran Boral, Dina Friedland, and W. Kevin Wilkinson for their contributions to this paper especially their help in deriving the execution time of the projection operation in the presence of duplicates.

7. References

[ASTR76] Astrahan, M.M., et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

[BABB79] Babb,E. "Implementing a Relational Database by Means of Specialized Hardware," ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979, pp. 1-29.

[BANC80] Bancilhon P. and M. Scholl, "Design of a Backend Processor for a Data Base Machine," Proc. of the ACM SIGMOD 1980 International Conference of Management of Data, May 1980.

[BANE78] Banerjee J., R.I. Baum, and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," ACM TODS, Vol. 3, No. 4, Dec. 1978.

[BLAS77] Blasgen M.W. and K.P. Eswaran, "Storage and Access in Relational Data Bases", IBM System Journal, Vol. 16, No. 4, 1977.

[BORA80] Boral, H., DeWitt, D. J., Friedland, D., and W. K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," submitted to ACM Transactions on Database Systems, October, 1980.

[BORA81] Boral H., D.J. DeWitt, and W.K. Wilkinson, "Performance Evaluation of Four Associative Disk Designs," submitted to the Journal of Information Systems, March 1981.

[CHAM81] Chamberline, D.D., et. al., "Support for Repetitive Transactions and Ad Hoc Queries in System R," ACM Transactions on Database Systems, Vol. 6, No. 1, March, 1981.

[DEWI79] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June 1979, pp. 395-406.

[GORS80] Gorsline G.W., Computer Organization: Hardware/Software, Prentice-Hall, 1980, p. 149.

[GRAY78] Gray,J.N., "Notes on Database Operating Systems", Report RJ2188, IBM Research, San Jose, California, 1978.

[GRIP79] Griffiths Selinger P., M.M. Astrahan, D.D. Chamberline, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," Proc. of the ACM SIGMOD 1979 International Conference of Management of Data, May 1979.

[HAWT79] Hawthorn, Paula, "Evaluation and Enhancement of the Performance of Relational Database Management Systems,"

Electronics Research Laboratory, University of California at Berkeley, Memo No. M79-70.

[HANT80] Hawthorn P. and D.J. DeWitt, "Performance Evaluation of Database Machines," To appear IEEE Transactions on Software Engineering.

[HELL81] Hell, W., "RDBM - A Relational Data Base Machine: Architecture and Hardware Design," Proceedings of the 6th Workshop on Computer Architecture for Non-Numeric Processing, June 1981.

[MADN79] Madnick S.E., "The INFOPLEX Database Computer: Concepts and Directions," Proc. IEEE Computer Conference, Feb. 1979.

[KANN78] Kannan, Krishnamurthi, "The Design of a Mass Memory for a Database Computer," Proc. Fifth Annual Symposium on Computer Architecture, Palo Alto, CA. April 1978.

[KIBL80] Kibler T.R., "APCAM - A Practical Cellular Associative Memory," Fifth Workshop on Computer Arch. for Non-numeric Processing, Mar., 1980.

[KNUT75] Knuth D.E., The Art of Computer Programming - Sorting and Searching Addison-Wesley, 1975, p. 160.

[LEIL78] Leilich H.O., G. Stiege, and H.Ch. Zeidler, "A Search Processor for Data Base Management Systems," Proc. 4th Conference on Very Large Databases, 1978.

[LIN76] Lin, S.C., D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," TODS vol. 1, No. 1, pages 53 - 75, Mar. 1976.

[MINS72] Minsky M., "Rotating Storage Devices as Partially Associative Memories," Proc. 1972 FJCC.

[OZKA75] Ozkarahan, E.A., S.A. Schuster, and K.C. Smith, "RAP - Associative Processor for Database Management," AFIPS Conference Proceedings, vol. 44, 1975, pp. 379 - 388.

[OZKA77] Ozkarahan, E.A., Schuster, S.A. and Sevcik, K.C., "Performance Evaluation of a Relational Associative Processor," ACM Transactions on Database Systems, Vol. 2, No.2, June 1977. Communications ACM 17, 7 , July,1974.

[PARH72] Parhami B., "A Highly Parallel Computing System for Information Retrieval," Proc. 1972 FJCC.

[PARK71] Parker J.L., "A Logic per Track retrieval System," IFIP Congress, 1971.

[SADO78] Sadowski P.J. and S.A. Schuster, "Exploiting Parallelism

in a Relational Associative Processor," Fourth Workshop on Computer Arch. for Non-numeric Processing, Aug. 1978.

[SLOT70] Slotnik, D.L. "Logic per Track Devices" in "Advances in Computers", Vol. 10., Frantz Alt, Ed. Academic Press, New York, 1970, pp 291 - 296.

[STON76] Stonebraker, M. R. et. al., "The Design and Implementation of INGRES," TODS, Vol 1, No. 3, September 1976.

[STON79] Stonebraker, M. R., "MUFFIN: A Distributed Database Machine," University of California, Electronics Research Laboratory, Memo UCB/ERL m79/28, May, 1979.

[SU75] Su, Stanley Y. W., and G. Jack Lipovski, "CASSM: A Cellular System for Very Large Data Bases", Proceedings of the VLDB, 1975, pages 456 - 472.

[UPCH79] Upchurch E.T., J.R. Bitner, D.P.S Charlu, and A.G. Dale, "A Reconfigurable Database Machine: Architecture and Algorithms," Institute for Computer Science and Computer Applications, The University of Texas at Austin, March 1979.

[WONG76] Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing," ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 223-241.

8. Appendix I - Execution time for Projection Operation

Assumptions:
Relation starts with R blocks
Projected relation contains N blocks where N<R
The merge factor utilized is Z

The algorithm is broken into two phases. Phase 1 starts with R runs of length 1 and produces R/N runs of length N. Since we are developing an upper bound on the execution time of the operation, we assume that in forming the R/N runs of length N no duplicates are found. The first phase has $\log_Z(N)$ stages with run lengths of 1, Z, Z^2, ... , N blocks respectively. Since no duplicates are found (or eliminated) during this phase, all R pages are read and written during each stage. Thus, each of the $\log_Z(N)$ stages requires:

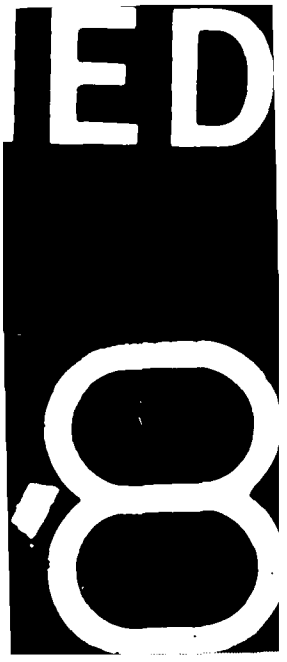
$$2*T_{DAC} + 2*(\lceil R/19 \rceil - 1)*T_{SK} + 2*R*(T_{IO}+T_{OIO}) + R*T_{BLK} \text{ ms.}$$

The second phase starts with R/N runs of length N and terminates with one run of length N using a Z way merge. Assuming that duplicates are uniformly distributed and that the number of blocks is reduced by a factor of Z at each stage, then this phase has $k=\log_Z(R/N)$ stages. Thus, $Z^k = R/N$. During the first stage Z^k*N pages are read and processed and $Z^{k-1}*N$ pages are written. In the second stage $Z^{k-1}*n$ pages are read and processed and $Z^{k-2}*N$ pages are written. The k th. stage begins by reading and processing Z^1*N pages and ends by writing Z^0*N pages. Summing the operations performed during the k stages yields:

$$(R-N)/(Z-1)*\{Z*(T_{IO}+T_{OIO}+T_{BLK}+T_{SK}/19) + (T_{IO}+T_{OIO}+T_{SK}/19)\}$$
$$+ \log_Z(R/N)*2*T_{DAC}$$

We have assumed that one of every 19 pages read or written

requires a track-to-track seek and that every stage requires one random disk access to begin reading blocks and one to begin writing blocks.


ED
'8